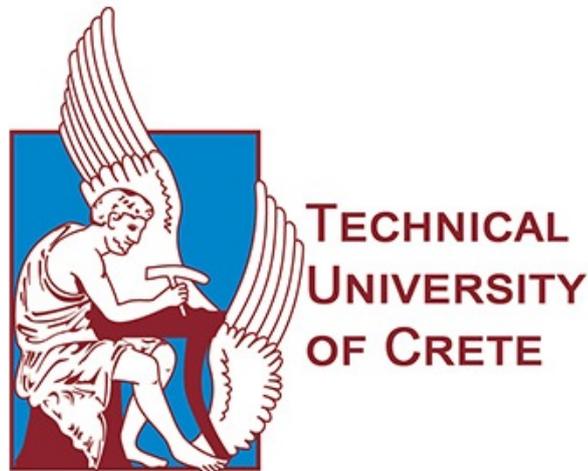


Technical University of Crete
School of Electrical and Computer Engineering



Implementation of ARM processor by using Bluespec language

Pekridis Georgios

Thesis Committee
Professor Dionisios Pnevmatikatos (Supervisor)(ECE)
Professor Apostolos Dollas (ECE)
Associate Professor Ioannis Papaefstathiou (ECE)

Chania, January 2018

Abstract

The goal of this thesis was to construct an ARM processor using the Bluespec System Verilog language(BSV). BSV has a fundamentally different approach to hardware design, comparing to other Hardware Description Languages. It is based on circuit generation rather than merely circuit description and is also based on atomic transactional rules instead of a globally synchronous view of the world. The processor belongs to the ARM7 family, it has a 3-stage pipeline, it uses a 32-bit architecture and is based on ARMv4 instruction set. In addition the processor supports all the operating modes. The modes of operation are User, Fast Interrupt(FIQ), Interrupt(IRQ), Supervisor, Abort, System and Undefined. The amount of different types of instructions that were implemented is 26. Each and every one of these types has additional functions depending on the condition codes and the addressing modes of the instruction. For the verification of the design, assembly code was used. This assembly code was produced by C++ code, through the ARM GCC.

Acknowledgements

First of all I would like to thank my supervisor Prof. Dionisios Pnevmatikatos for his guidance and support throughout this work and for giving me the opportunity to work on this very interesting topic.

I would also like to express my gratitude to Prof. Apostolos Dollas and Prof. Yannis Papaef-tathiou of being members of the committee and for evaluating my thesis.

Furthermore I would also to thank my friends that supported me all these years during my studies in Chania.

Last but most important I would like to thank my family and my girlfriend for supporting me in each step of the way, all these past years.

Contents

Abstract	i
Acknowledgements	iii
1 Introduction	1
2 Bluespec System Verilog	3
2.1 Bluespec Syntax	4
2.2 Data types in Bluespec	5
2.3 The Bluespec compiler	7
2.3.1 Scheduling	8
2.3.2 The Bluesim simulator	10
3 ARM	11
3.1 About the ARM Architecture	11
3.1.1 Processor modes	12
3.1.2 ARM registers	13
3.1.3 Program status registers (PSR)	15
3.2 ARMv4 instruction set	17
3.2.1 The condition field	17
3.2.2 The barrel shifter	18
3.2.3 Branch instructions	19
3.2.4 Data processing instructions	19
3.2.5 Multiply instructions	20

3.2.6	Status register access instructions	21
3.2.7	Load and store instructions	21
4	Implementation	24
4.1	Register file	25
4.2	Decode Instruction	26
4.3	Barrel shifter	27
4.4	Execution module	28
4.4.1	ALU function	28
4.4.2	checkFlags function	28
4.4.3	Multiply signed and unsigned	28
4.4.4	brAddrCalc function	29
4.4.5	Execute function	29
4.5	Instruction and Data memory	30
4.6	Coprocessor	30
4.7	Processor module	30
4.7.1	preFetch rule	31
4.7.2	doFetch rule	31
4.7.3	doDecode rule	32
4.7.4	doExecute rule	32
4.7.5	doWriteBack rule	36
4.7.6	doLongMul rule	36
5	Debugging and Testing	37
5.1	Debug of the design	37
5.1.1	Trace module	37
5.2	Testing of the register file	38
5.3	Testing the barrel shifter	40
5.4	Testing the processor	41
5.4.1	Linear search	43

5.4.2	Largest number among 3	45
5.4.3	Factorial	46
5.4.4	Fibonacci	48
5.4.5	Bubble Sort	50
5.5	Compare with LEON and Piccolo	52
6	Conclusion	54
6.1	Conclusion of Thesis	54
6.2	Future Work	55
	Bibliography	55

List of Tables

3.1	Processor modes	12
3.2	Condition codes	18
3.3	Data-processing instructions	20
5.1	LEON-Piccolo-Our design overview	53
5.2	Detailed allocation of resources	53

List of Figures

2.1	A Bluespec module	3
2.2	Diagram of Bluespec's compiler flow	8
3.1	The ARM Register Set	14
3.2	The PSR	15
3.3	Instruction set encoding	17
3.4	ARM Datapath	23
4.1	ARM 3-stage pipeline	24
4.2	Block diagram of the design	25
5.1	TestBench the register file	39
5.2	TestBench VCD	40
5.3	Barrel Shifter TestBench	40
5.4	Barrel Shifter console output	40
5.5	Linear Search console output	44
5.6	Linear Search VCD	44
5.7	Largest number among 3 C++ and assembly code	45
5.8	Largest number among 3 VCD	45
5.9	Factorial C++ and assembly code	46
5.10	Factorial console output	47
5.11	Factorial VCD output	47
5.12	Fibonacci sequence C++ and assembly code	48
5.13	Fibonacci sequence console output	49

5.14 Fibonacci sequence VCD output	49
5.15 Bubble sort C++ code	50
5.16 Bubble short assembly code part 1 Image	51
5.17 Bubble short assembly code part 2	51
5.18 Bubble short assembly code part 3	51
5.19 Bubble sort VCD	51

Chapter 1

Introduction

The need to speed up the hardware design cycle has caused industry to look at more powerful tools for hardware synthesis from high-level descriptions. One of these tools is Bluespec. Bluespec is a strongly-typed hardware synthesis language which makes use of the Term Rewriting System(TRS) to describe computation as a series of atomic state changes.

Bluespec has been used at many Universities around the world for academic and research purposes. Also there is a commercial RISC-V processor core made in bluespec language named Piccolo. Projects and courses[5] in other Universities, such as MIT, shown that a simple processor model like MIPS can be done quite efficiently. In addition the study[7] has shown that the RTL Verilog generated by Bluespec compiler is often better or equivalent to hand-coded RTL Verilog. So if there is a tool powerful enough to help developers to make advanced and complex designs more easily, then why not to take advantage of it.

In this work we explore the benefits of Bluespec System Verilog in hardware design implementing a 3-stage pipeline ARM processor core using ARMv4 instruction set architecture(ISA). Our work will be compared with other processors as far as it concern the clock-speed and the consumption of resources on a FPGA. The processor supports the seven operating modes that ARMv4 suggests. An equally important factor of our work was the verification of the design. So in order to do that, we used programs written in C++ which were translated to assembly via the ARM GCC. As a last step of transforming the code to binary instructions, we took

advantage of the GNU Embedded Toolchain for ARM. With the use of this tool, binary files was taken as outputs and these files was loaded to the Instruction memory to check the function of our processor.

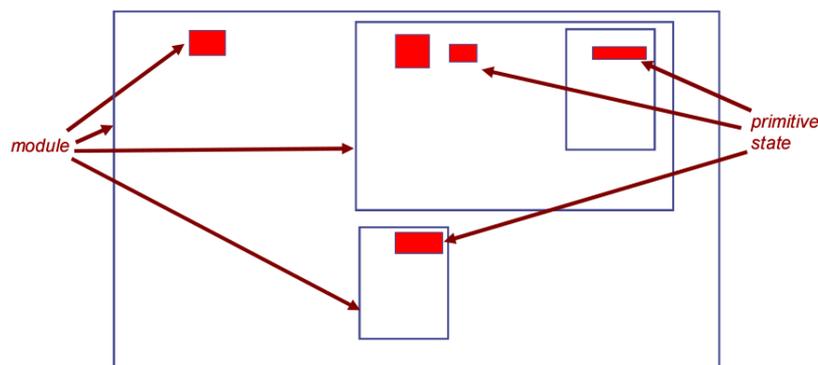
Chapter 2

Bluespec System Verilog

Bluespec is a hardware description language (HDL) that compiles into TRS. This intermediate TRS description can then be translated through a compiler into either in Verilog RTL or a cycle-accurate C-simulation.

A general circuit is represented as a module in Bluespec. The produced object is that which is compiled to RTL. There is a correspondence in Bluespec and Verilog modules. Three elements compose each module. These are the state, the rules and the interface. The state can be described from registers, flip-flops and memories. The rules are actions that modify that state. Last, interfaces which provide a mechanism for interaction of the external environment with the internal structure of the module.

Figure 2.1: A Bluespec module



2.1 Bluespec Syntax

Initially Bluespec System Verilog design consists of a module hierarchy (just like in Verilog, SystemVerilog and SystemC). The leaves of the hierarchy are primitive state elements, including registers, FIFOs, etc. Even registers are (semantically) modules (unlike in Verilog, SystemVerilog). The behavior of a module is represented by its rules each of which consists of a state change on the hardware state of the module (*an action*) and the conditions required for the rule to be valid (*a predicate*). It is valid to execute (*fire*) a rule whenever its predicate is true. The syntax for a rule is:

```
rule ruleName[( condition)];
actions
endrule [:ruleName]
```

As it said before every module has an interface. The interface of a module is a set of methods through which the outside world interacts with the module. Each interface method has a predicate (a guard) which restricts when the method may be called. A method may either be a read method (a combinational lookup returning a value), an action method, or a combination of the two, an actionvalue method. An actionvalue is used when we do not want a combinational lookups result to be made available unless an appropriate action in the module also occurs. The syntax of the interface is:

```
interface IfcName[ #( ifc type params) ];
method type methodName (type arg, ..., type arg);
...
method type methodName (type arg, ..., type arg);
```

endinterface [: IfcName]

There are three things to take into consideration for a rule to fire. First of all is the rule's condition. If this condition is false, the rule does not fire. If there is no condition then the rule can fire in every clock cycle. Secondly, the methods have "ready" signals. Ready signals are specified for each method in defining module. Rule does not fire unless all ready conditions are true. Finally a rule may not fire because it conflicts with other rules. Conflict of rules means that the compiler needs to decide which rule have to fire first. A conflict of rules is created in the case where two or more different rules affect the state of a module in the same clock cycle.

2.2 Data types in Bluespec

Bluespec provides a strong, static type-checking environment. Every variable and every expression has a type. Variables must be assigned values which have compatible types. Type checking, which occurs before program elaboration or execution, ensures that object types are compatible and that needed conversion functions are valid for the context.

Common types One way to classify types in Bluespec are whether they are in the Bits class. Bits defines the class of types that can be converted to bit vectors and back. Only types in the Bits class are synthesizable and can be stored in a state element, such as a Register or a FIFO.

- **Bit Types**

- *Bool*: True or False values.
- *Bit#(n)*: n Bits
- *UInt#(n)*: Unsigned fixed width (n) representation of an integer value
- *Int#(n)*: Signed fixed width (n) representation of an integer value

- **Non Bit Types**

- *Integer* : Integers are unbounded in size and are commonly used as loop indices for compile-time evaluation.
- *String* : Strings are mostly used in system functions (such as `$display`). They can be tested for equality and inequality.
- *Interface* : Since interfaces are considered a type they can be passed to and returned from functions.
- Also types are :
 - * Action
 - * ActionValue
 - * Rules
 - * Modules
 - * Functions

- **User Defined Types**

- *Enum*: Similar to most languages, you can define names to be used in your code. Enum labels must all start with an uppercase letter.
- *Struct* : Structures contain members. A struct value contains the first member and the second member and the third member, and so on. Structure member names begin with a lowercase letter.
- *Tagged Union*: Tagged unions also contain members. A tagged union value contains the first member or the second member or the third member, and so on. Tagged union member names begin with a lowercase letter.

- **Types from the Bluespec Library**

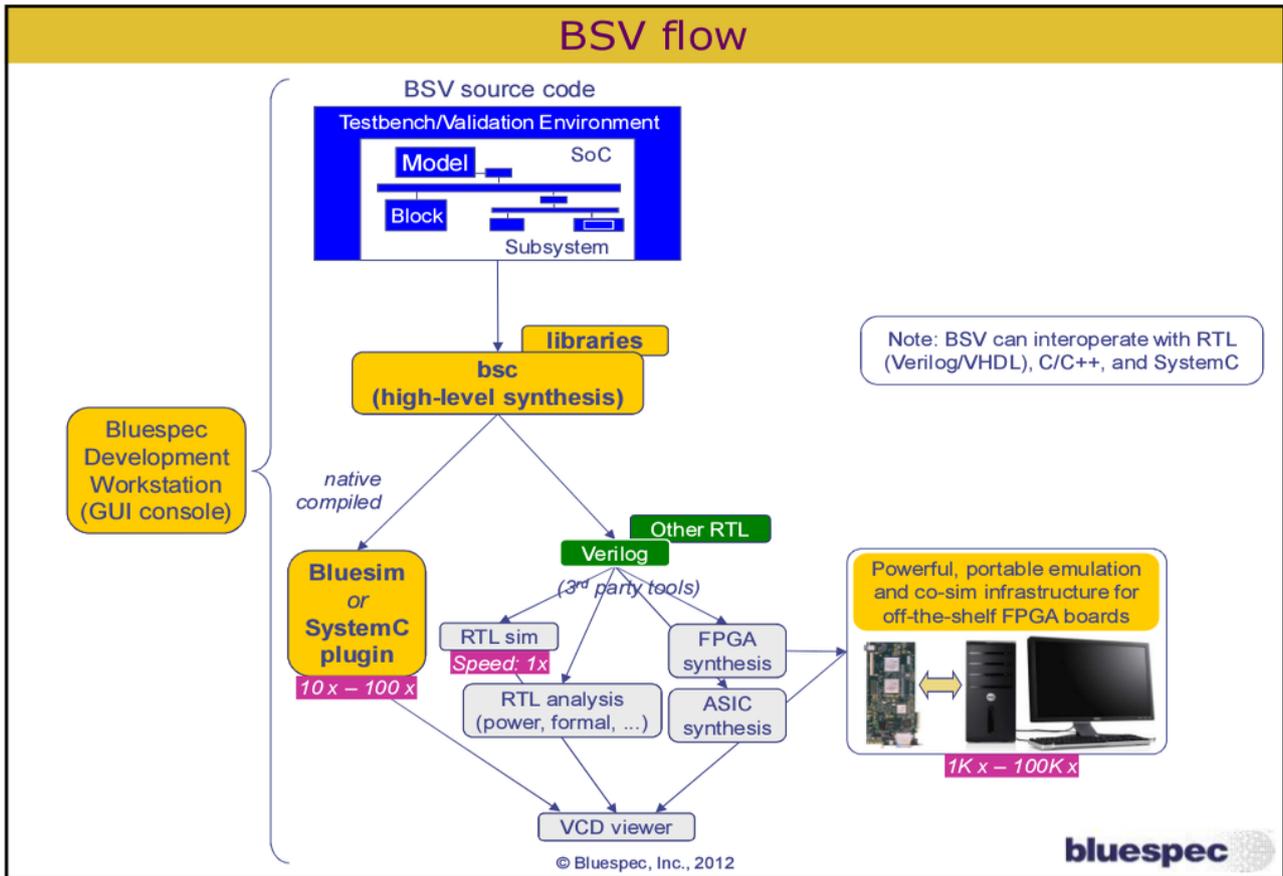
- *Maybe*: The *Maybe* type encapsulates any data type `a` with a valid bit and further provides some functions to test the valid bit and extract the data.
- *Vector*: A *Vector* is a container of a specific length holding elements of one type. To use this type the *Vector* package must be imported.

- *Tuple*: A *Tuple* provides an unnamed data structure typically used to hold a small number of related fields. Like other SystemVerilog constructs, tuples are polymorphic, that is, they are parameterized on the types of data they hold. Tuples are typically found as return values from functions when more than one value must be returned.

2.3 The Bluespec compiler

The Bluespec compiler can translate Bluespec descriptions into either Verilog RTL or a cycle-accurate SystemC simulation (Figure 2-2). It does so by initially evaluating the high-level description of the design into a TRS description of rules and state. From this TRS description the compiler schedules the actions and transforms the design into a timing-aware hardware description. This task involves determining when rules can fire safely and concurrently, adding MUXes logic to handle the sharing of state elements by rules, and finally applying boolean optimizations to simplify the design. From this timing-aware model, the compiler can produce a synthesizable Verilog RTL or SystemC executable output .

Figure 2.2: Diagram of Bluespec's compiler flow



2.3.1 Scheduling

Scheduling is the task of determining what subset of rules should fire on a cycle, given its state and in what order should rules be fired in a that very cycle. It is essential for someone who desires to use Bluespec proficiently, to firstly comprehend the mechanisms of Bluespec and how its compiler schedules multiple rules for cycle-by-cycle execution.

Determining rule contents

Due to the complexity of determining when a rule will use an interface of a module, the Bluespec compiler assumes conservatively that an action will use any method possible from the aforementioned interface. Henceforth, if an action uses a method only when some condition is met, the scheduler will treat it as if were always using it. This leads the compiler to make conservative estimations of method usage which in turn causes conservative firing conditions to be scheduled.

Determining Pair-wise Scheduling Conflicts

Once the components (methods and other actions) of all the actions have been determined, all possible conflicts, between each atomic action pair, have to be discovered. In the case that two rules' predicates are provably disjoint, then it can be assumed that there are no conflicts as they can never happen in the same cycle. Otherwise, the scheduling conflicts between them is exactly the set of scheduling conflicts between any pair of action components of each atomic action.

For example, consider rules *rule1* and *rule2* where *rule1* reads some register *r1* and *rule2* writes it. Registers have the scheduling constraint *read*<*write*, which means that calls to the read method calls must happen before the write method call in a single cycle. Thus this constraint is reflected in the constraints between *rule1* and *rule2* (*rule1* < *rule2*). If *rule1* were to also write some register *r2* and *rule2* were to read it we would have the additional constraint (*rule2* < *rule1*). In this there is no consistent way of ordering the two rules, so we consider the rules conflicting with sequential ordering restrictions (as they will never happen together, it doesn't matter how they are ordered to happen concurrently).

Generating a Final Global Schedule

Once all the pair-wise conflicts between actions have been determined, a temporal ordering of the actions takes place. For this to happen, the compiler orders the atomic transactions by some metric of importance, which is called urgency. Scheduler sorts each action, in descending urgency order. The goal is to place the action in a position that prevents the most conflicts, with already ordered rules, in this process. Only when its ordering has been determined, the rule is allowed to be fired in a cycle, when respectively its predicate is met and there are no more urgent rules, which conflict with it in that total ordering. Once the compiler has considered all atomic transactions in sequence, we have a complete schedule.

2.3.2 The Bluesim simulator

Bluesim delivers high-speed simulation of BSV designs at a source-level or with SystemC executables. Bluesim can be at least 10x faster than the standard Verilog simulator. The main features of the simulator are that it has high-speed and the output of a BSV high-level-design is a source-level or SystemC executable simulation. Also Bluesim is 100% cycle accurate with Verilog RTL and it generates standard VCD files. Therefore the benefits of these are that the simulation can be accelerated as well as the verification of the design.

Chapter 3

ARM

Our implementation of ARM is based on the ARM7 family of processors. Our processor has a 32-bit architecture, 3-stage pipeline and is based on ARMv4 instruction set. In the section below will be analyzed the ARMv4 instructions that were implemented in our design

3.1 About the ARM Architecture

The ARM is a Reduced Instruction Set Computer (RISC), as it incorporates these typical RISC architecture features:

- a large uniform register file
- a load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only
- uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture provides:

- control over both the Arithmetic Logic Unit (ALU) and shifter in most data-processing instructions to maximize the use of an ALU and a shifter
- auto-increment and auto-decrement addressing modes to optimize program loops

- Load and Store Multiple instructions to maximize data throughput instruction fields only
- conditional execution of almost all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

3.1.1 Processor modes

The ARM architecture supports the seven processor modes shown in Table 3.1.

Table 3.1: Processor modes

Processor	mode	Mode number	Description
User	usr	5b10000	Normal program execution mode
FIQ	fiq	5b10001	Supports a high-speed data transfer or channel process
IRQ	irq	5b10010	Used for general-purpose interrupt handling
Supervisor	svc	5b10011	A protected mode for the operating system
Abort	abt	5b10111	Implement virtual memory and/or memory protection
Undefined	und	5b11011	Supports software emulation of hardware coprocessors
System	sys	5b11111	Runs privileged operating system tasks(ARMv4 and above)

Most application programs execute in User mode. When the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur. This allows a suitably-written operating system to control the use of system resources.

The modes other than User mode are known as privileged modes. They have full access to system resources and can change mode freely. Five of them are known as exception modes:

- FIQ
- IRQ
- Supervisor
- Abort
- Undefined

The remaining mode is System mode, which is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks that need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.

3.1.2 ARM registers

ARM has 37 , 32-bit registers in total.

- 1 dedicated program counter
- 1 dedicated current program status register
- 5 dedicated saved program status registers
- 30 general purpose registers

At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers. The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged, which means:

- User mode can only switch to another processor mode by generating an exception. The SWI instruction provides this facility from program control.
- Memory systems and co-processors might allow User mode less access to memory and co-processor functionality than a privileged mode.

Three of 16 visible register have special roles:

Stack pointer Software normally uses R13 as a Stack Pointer (SP).

Link register Register 14 is the Link Register (LR). This register holds the address of the next instruction after a Branch and Link (BL) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.

Program counter Register 15 is the Program Counter (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. In ARM state, all ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.

The remaining 13 registers have no special hardware purpose. Their uses are defined purely by software. The visible registers of each operating mode are shown in the Figure 3.1

Figure 3.1: The ARM Register Set

Modes						
	Privileged modes					
	Exception modes					
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Z Is set to 1 if the result of the instruction is zero (this often indicates an equal result from a comparison), and to 0 otherwise.

C Is set in one of four ways:

- For an addition, including the comparison instruction `CMN`, **C** is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, **C** is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, **C** is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, **C** is normally left unchanged (but see the individual instruction descriptions for any special cases).

V Is set in one of two ways:

- For an addition or subtraction, **V** is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
- For non-addition/subtractions, **V** is normally left unchanged (but see the individual instruction descriptions for any special cases).

The flags can be modified in these additional ways:

- Execution of an `MSR` instruction, as part of its function of writing a new value to the `CPSR` or `SPSR`
- Execution of `MRC` instructions with destination register `R15`. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.
- Execution of some variants of the `LDM` instruction. These variants copy the `SPSR` to the `CPSR`, and their main intended use is for returning from exceptions.

- Execution of an RFE instruction in a privileged mode that loads a new value into the CPSR from memory.
- Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are intended for returning from exceptions.

The mode bits

The M[4:0] are the mode bits. These determine the mode in which the processor operates. Their interpretation is shown in Table 3.1 .

3.2 ARMv4 instruction set

The figure below presents the instruction set encoding In the sections below are described only the instructions that our processor can execute.

Figure 3.3: Instruction set encoding

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Data processing immediate shift	cond	0	0	0	opcode			S	Rn				Rd				shift amount			shift	0	Rm														
Miscellaneous instructions:	cond	0	0	0	1	0	x	x	0	x																	0	x								
Data processing register shift	cond	0	0	0	opcode			S	Rn				Rd				Rs			0	shift	1	Rm													
Miscellaneous instructions:	cond	0	0	0	1	0	x	x	0	x																	0	x	x	1	x					
Multiplies: Extra load/stores:	cond	0	0	0	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	1	x							
Data processing immediate	cond	0	0	1	opcode			S	Rn				Rd				rotate			immediate																
Undefined instruction	cond	0	0	1	1	0	x	0	0	x																										
Move immediate to status register	cond	0	0	1	1	0	R	1	0	Mask				SBO				rotate			immediate															
Load/store immediate offset	cond	0	1	0	P	U	B	W	L	Rn				Rd				immediate																		
Load/store register offset	cond	0	1	1	P	U	B	W	L	Rn				Rd				shift amount			shift	0	Rm													
Media instructions	cond	0	1	1	x				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x						
Architecturally undefined	cond	0	1	1	1	1	1	1	1	x																	1	1	1	1	x					
Load/store multiple	cond	1	0	0	P	U	S	W	L	Rn				register list																						
Branch and branch with link	cond	1	0	1	L	24-bit offset																														
Coprocessor load/store and double register transfers	cond	1	1	0	P	U	N	W	L	Rn				CRd				cp_num			8-bit offset															
Coprocessor data processing	cond	1	1	1	0	opcode1			CRn				CRd				cp_num			opcode2	0	CRm														
Coprocessor register transfers	cond	1	1	1	0	opcode1			L	CRn				Rd				cp_num			opcode2	1	CRm													
Software interrupt	cond	1	1	1	1	swi number																														
Unconditional instructions:	1	1	1	1	x																															

3.2.1 The condition field

Most ARM instructions can be conditionally executed, which means that they only have their normal effect on the programmers model state, memory and coprocessors if the N, Z, C

and V flags in the CPSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP: that is, execution advances to the next instruction as normal, including any relevant checks for interrupts and Prefetch Aborts, but has no other effect. Every instruction contains a 4-bit condition code field in bits 31 to 28.

Table 3.2: Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear(N==V)
1011	LT	Signed less than	N set and V clear, or N clear and V set(N!=V)
1100	GT	Signed greater than	Z clear and either N set and V set, or N clear and V clear(Z==0,N==V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set(Z ==1 or N!=V)
1110	AL	Always(uncoditional)	-
1111	-	-	-

3.2.2 The barrel shifter

The ARM does not have actual shift instruction. Instead it has a barrel shifter which provides a mechanism to carry out shifts as a part of other instructions. The operations that barrel shifter supports are:

- LSL: Logical shift left
- LSR: Logical shift right
- ASR: Arithmetic Shift Right

- Shifts right and preserves the sign bit for 2's complement operations
- ROR: Rotate Right
- RRX: Rotate with extend
 - This operation uses the CPSR C flag as a 33rd bit. Rotates right by 1 bit.

3.2.3 Branch instructions

All ARM processors support a branch instruction that allows a conditional branch forwards or backwards up to 32MB. As the PC is one of the general-purpose registers (R15), a branch or jump can also be generated by writing a value to R15. A subroutine call can be performed by a variant of the standard branch instruction. As well as allowing a branch forward or backward up to 32MB, the Branch with Link (BL) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14). When executing the instruction, the processor shifts the offset left two bits, sign extends it to 32 bits and adds it to PC. Execution then continues from the new PC, once the pipeline has been refilled. The processor wants 3 cycles to refill the pipeline.

3.2.4 Data processing instructions

ARM has 16 data-processing instructions shown in Table below

Table 3.3: Data-processing instructions

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd:=Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd:=Rn OR shifter_operand
0010	SUB	Subtract	Rd:=Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd:=shifter_operand-Rn
0100	ADD	Add	Rd:=Rn + shifter_operand
0101	ADC	Add with Carry	Rd:=Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd:=Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd:=shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive)OR	Rd:=Rn OR shifter_operand
1101	MOV	Move	Rd:=shifter_operand(no first operand)
1110	BIC	Bit Clear	Rd:=Rn AND NOT (shifter_operand)
1111	MVN	Move Not	Rd:=NOT shifter_operand(no first operand)

3.2.5 Multiply instructions

ARMv4 has 2 classes of Multiply instruction:

Normal 32-bit x 32-bit, bottom 32-bit result

Long 32-bit x 32-bit, bottom 64-bit result

Normal multiply

There are two 32-bit x 32-bit Multiply instructions that produce bottom 32-bit results:

MUL Multiplies the values of two registers together, truncates the result to 32 bits, and stores the result in a third register.

MLA Multiplies the values of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result in a fourth register. This can be used to perform multiply-accumulate operations.

Both Normal Multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags. No distinction is made between signed and unsigned variants. Only the least

significant 32 bits of the result are stored in the destination register, and the sign of the operands does not affect this value.

Long multiply

There are five 32-bit x 32-bit Multiply instructions that produce 64-bit results. Two of the variants multiply the values of two registers together and store the 64-bit result in third and fourth registers. There are signed (**SMULL**) and unsigned (**UMULL**) variants. The signed variants produce a different result in the most significant 32 bits if either or both of the source operands is negative. Two variants multiply the values of two registers together, add the 64-bit value from the third and fourth registers, and store the 64-bit result back into those registers (third and fourth). There are signed (**SMLAL**) and unsigned (**UMLAL**) variants. These instructions perform a long multiply and accumulate. All the Long Multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags.

3.2.6 Status register access instructions

There are two instructions for moving the contents of a program status register to or from a general-purpose register. Both the CPSR and SPSR can be accessed.

MRS Move PSR to General-purpose Register.

MSR Move General-purpose Register to PSR

3.2.7 Load and store instructions

The ARM architecture supports two broad types of instruction which load or store the value of a single register, or a pair of registers, from or to memory:

- The first type can load or store a 32-bit word or an 8-bit unsigned byte.
- The second type can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte.

Addressing modes

In both types of instruction, the addressing mode is formed from two parts, the base register and the offset. The base register can be any one of the general-purpose registers (including the PC, which allows PC-relative addressing for position-independent code). The offset takes one of three formats:

Immediate The offset is an unsigned number that can be added to or subtracted from the base register. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers. For the word and unsigned byte instructions, the immediate offset is a 12-bit number. For the halfword and signed byte instructions, it is an 8-bit number.

Register The offset is a general-purpose register (not the PC), that can be added to or subtracted from the base register. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register (not the PC) shifted by an immediate value, then added to or subtracted from the base register. The same shift operations used for data-processing instructions can be used (Logical Shift Left, Logical Shift Right, Arithmetic Shift Right and Rotate Right), but Logical Shift Left is the most useful as it allows an array indexed to be scaled by the size of each array element. Scaled register offsets are only available for the word and unsigned byte instructions.

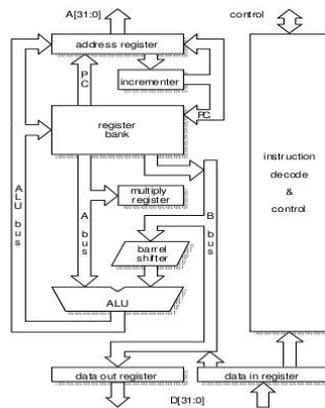
As well as the three types of offset, the offset and base register are used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The base register and offset are added or subtracted to form the memory address.

Pre-indexed The base register and offset are added or subtracted to form the memory address. The base register is then updated with this new address, to allow automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The base register and offset are added or subtracted and this value is stored back in the base register, to allow automatic indexing through an array or memory block. The datapath of ARM7 is as follows.

Figure 3.4: ARM Datapath

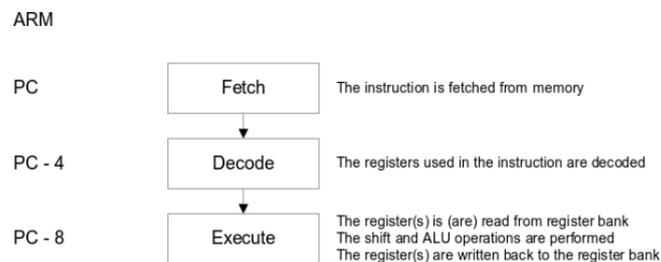


Chapter 4

Implementation

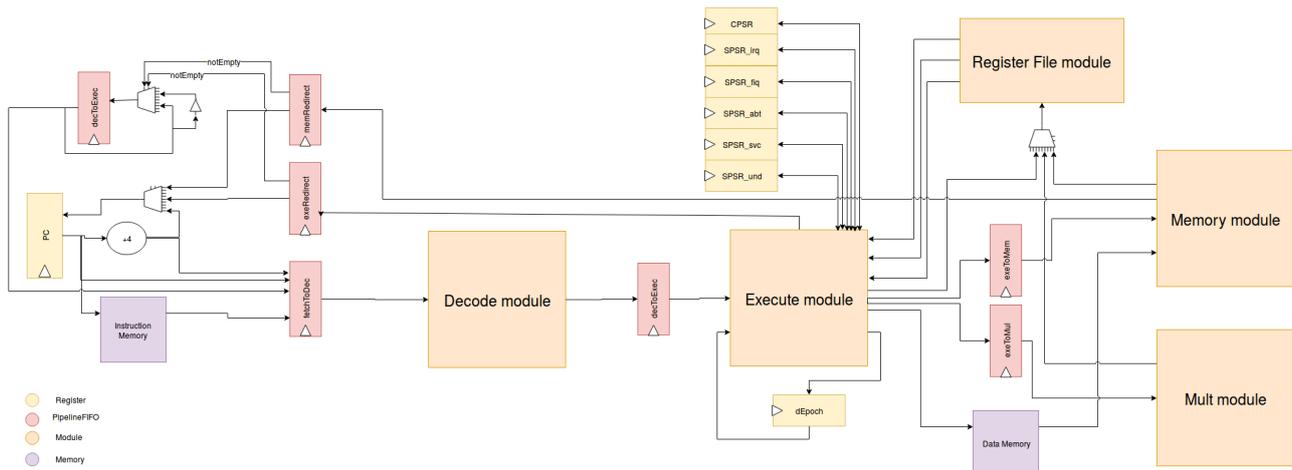
In this module we will analyze the structural components of the processor. Due to the fact that there is not much literature or examples on creating processors with bluespec, we decided to implement our design on the basis of the book[5]. Our design is a 3 stage pipeline processor. The three stages are Fetch, Decode and Execute.

Figure 4.1: ARM 3-stage pipeline



Except the basic stages, there are 2 more that deals with the instructions that demand one more clock cycle to complete. Our design implements all Data processing instructions as they are described in Table 3.3, Branch and Branch and Link instructions, Load and Store instructions with offset indexed addressing, post and pre indexed addressing and six instructions of multiplication. These instruction are signed and unsigned multiplication with or without accumulation and the product that is stored may be 32 or 64 bit(long multiplication). Additionally the processor supports all the operating modes of the ARMv4 ISA. A general block diagram of our architecture is shown below.

Figure 4.2: Block diagram of the design



4.1 Register file

The register file of the processor is composed from vectors of registers. Registers in Bluespec can store any type of data like integers, bits, strings even whole structures of data. In our case the registers of the register file stores 32-bit. There are six vectors of registers in total, one for each operating mode of the processor that uses different registers than the user mode. PC, CPSR and SPSR registers for each mode are not inside of the register file, but there are implemented on the top module of the processor for convenience. The interface of the module has one method for writing and three methods for reading. The location of reading or writing the data depends on the index and the mode, that are given to the methods as arguments. Below are shown the interface and the vectors of the register file.

```

interface RFile;
method Action wr(Ridx ridx, Bit#(32) data,CpuMode mode);
method Bit#(32) rd1(Ridx ridx,CpuMode mode);
method Bit#(32) rd2(Ridx ridx,CpuMode mode);
method Bit#(32) rd3(Ridx ridx,CpuMode mode);
endinterface

```

```

Vector#(15, Reg#(Data)) rfile <- replicateM(mkReg(0));

```

```

Vector #(2, Reg #(Data)) rBankSvc <- replicateM(mkReg(0));
Vector #(2, Reg #(Data)) rBankIrq <- replicateM(mkReg(0));
Vector #(2, Reg #(Data)) rBankAbt <- replicateM(mkReg(0));
Vector #(2, Reg #(Data)) rBankUnd <- replicateM(mkReg(0));
Vector #(7, Reg #(Data)) rBankFiq <- replicateM(mkReg(0));

```

4.2 Decode Instruction

The decoding of the instruction is achieved through a function. The function, based on the ARMv4 ISA, takes as input the 32-bit instruction and returns a structure. The format of the structure is:

```

typedef struct {
  IType iType;
  AluFunc aluFunc;
  Cond_flags cond;
  Maybe #(bit) set_flags;
  Maybe #(Ridx) rd;
  Maybe #(Ridx) rn;
  Maybe #(Ridx) rm;
  Maybe #(Rotate_imm) rot_imm;
  Maybe #(Data) imm;
  Maybe #(Ridx) rs;
  Maybe #(Bit #(2)) shift_type;
  Maybe #(Shift_imm) shift_length;
  Maybe #(bit) rorx;
  Maybe #(Bit #(24)) imm24;
  Maybe #(bit) pBit;

```

```

Maybe#(bit) wBit;
Maybe#(bit) bBit;
LdStType ldStType;

} DecodedInst deriving(Bits, Eq);

```

The IType field holds the type of the instruction, the AluFunc field holds the opcode for the ALU and the Cond.flags field is for the condition flags that exist in every instruction. The set.flags field implies if the instruction will change the CPSR flags. The fields rd, rn, rm, rs show which registers will be written or read. In addition the fields rot_imm, shift_type, shift_length, rorx affects the operation of the barrel shifter. The imm field represents the 11-bit immediate that the instruction might have. In the other hand the imm_24 field shows the 24-bit immediate that the Branch instructions have. Finally the pBit, wBit, bBit and ldStType defines the type of the load and store instructions

4.3 Barrel shifter

On the design barrel shifter was implemented as separate module. The syntax of the barrel shifter function is:

```

function BrlResult brlShifter(Bit #(2) control , Bit #(32) opperand, Bit #(5) rot_imm, Bit #(1)
carry, Bit #(1) rorx , Bit #(1) ibit);

```

The carry that the shifter takes as input is the C flag from the CPSR. For each value of the control argument the function performs a different action. The action of the shifter depends also on the ibit. If the ibit is '1' the function performs a left rotation on the operand by the amount that rot_imm indicates. Otherwise, the shifter operates as follows:

- `control==2'b00` : Logical Shift Left
- `control==2'b01` : Logical Shift Right
- `control==2'b10` : Arithmetic Shift Right
- `control==2'b11` : Rotate Bits Right

In the case of the latter if `rot_imm` and `rorx` bits are '0', then the shifter perform a rotate with extend. The output of the function is a 32-bit result and a 1-bit carry.

4.4 Execution module

The execution module of the processor is composed from some functions that are necessary for the execution of an instruction.

4.4.1 ALU function

First of all, there is the *alu* function that does all the logical and arithmetical operations, as mentioned in the Table 3.3. The input arguments of the function are the control of the ALU, the two operands and the carry flag fro CPSR. The output of *alu* function is a 32-bit result and the 4-bit flags (Negative,Zero,Carry,oVerflow). These flags are written in the CPSR if the *set_flags* bit is set.

4.4.2 checkFlags function

In continue, there is the *checkFlags* function that decides if an instruction has to execute or not. The decision shall be taken accordingly the condition flags of the instruction and the flags of the CPSR as described in the table 3.2.

4.4.3 Multiply signed and unsigned

There are two functions that do the multiply of two operands, one for signed and one for unsigned multiplication. The multiplication is made using the built-in function of Bluespec.

4.4.4 brAddrCalc function

This functions computes the new value of the PC, in case of a Branch or Branch and Link instruction. This function takes as input the address of the instruction, the type of the instruction, the 24-bit immediate of the instruction and a boolean value that comes from the checkFlags function. The new address for the PC is computed by the following expression :

$$\text{PC} + (\text{signExtend}(\text{imm24}) \text{ LSL by } 2)$$

4.4.5 Execute function

This function calls all the other functions that were mentioned before. Execute function takes as input arguments the value of the CPSR, the value of the SPSR of the current mode, the decoded instruction, the values that were read from the register file, the address of the instruction and the predicted address for the next instruction. The return value of the function is a structure. The fields of the structure is as it seems below:

```
typedef struct {
IType           iType;
Maybe#(Ridx)   rd;
Data            data;
Addr            addr;
Bool            mispredict;
Bool            brTaken;
Cond_flags      flags;
} ExecInst deriving(Bits, Eq);
```

Execute function distinguishes the instruction according on its type. Considering this separation, the fields of the structure are different from type to type. The iType field indicates the type of the Instruction (Data processing, Branch, Load, Store, Mul etc.). In addition The rd

shows if there is a destination register for the data to be stored. Furthermore the `addr` field indicates the address for reading or writing to the memory and the `brTaken` field is the result of the call of the function `checkFlags`. Also the `flags` field is the flags that the alu function has as an output. Finally `mispredict` field indicates that there is an unpredictable change in the contents of the PC, i.e. a different value than the expected `PC+4`. `Mispredict` field become true if the `rd` register is R15 or there is a B or BL instruction.

4.5 Instruction and Data memory

Instruction and Data memory are implemented as BRAMs. BRAMs are existing module inside the Bluespec's library. They have one port for reading and one for writing. In addition they have one cycle delay when writing or reading data and they can be initialize from a file.

4.6 Coprocessor

Coprocessor's module has limited functionality. It counts the clock cycles, receives the data that will be stored in `Rd` from the processor and displays certain messages purely for debugging.

4.7 Processor module

The main module of the design is the Processor module. Its interface consists of three methods. The first method is to receive the initial value of the PC from the coprocessor, another method changes the initial mode from which the processor will start the execution of the instructions and finally the third method is just an output for the data of the ALU. Every stage of the pipeline of the processor is written as a separate rule. There are six rules in total. In Bluespec the appropriate way to create pipeline for processors is to use FIFOs and in particular a `PipelineFIFO`. The reason of this is to take advantage the methods that FIFOs have in Bluespec and especially their *ready* signals. For example let's assume that there is a rule A that enqueues some data to a `PipelineFIFO`, that can store one element, and the rule B that wants to read this data and then dequeues it. If rule A does not enqueue the data to the FIFO then the

rule B can not fire because there is not something to dequeue. Respectively if rule B does not dequeue the data that was previously enqueued, then the rule A can not fire because the FIFO is full. So with this way we can synchronize the pipeline perfectly. Thus as we can easily understand if there are more rules that "communicate" with each other via PipelineFIFOs and one rule does not fire then the pipeline of the processor will be stalled. The main module also contains, besides the subcomponents that were mentioned in the other sections of this chapter, six FIFOs of different type, one register named *START* that has a boolean value to start and stop the processor and two more registers that hold a boolean value as well which indicates if the pipeline should stop because of a load or long multiplication or multiplication with accumulate instruction. These registers are named *pendingLoad* and *pendingLoadMul*. Furthermore there are two registers that control if an instruction should execute or not after a redirect of the PC.

4.7.1 preFetch rule

The preFetch rule fires if the *START* register is false and the coprocessor has started. This rule when fired, sends an request to the Instruction memory to read from the address that the PC indicates, increases the PC's value and changes the value of the *START* register to true. This rule fires once at the boot of the processor.

4.7.2 doFetch rule

This rule fires if the coprocessor has started the *START* register is true and there is no pending load due to a load or multiplication instruction. As the rule fires a check is being made whether there is a redirection of the PC. This redirection may have come either from the *doExecute* rule, either from the *doWriteBack* rule. In continue, the response of the Instruction memory is being read and if there is not a redirection for the PC then a new request is being made to the instruction memory. In case of a redirection, the request to the Instruction memory asks to read from the new value of the PC. As mentioned earlier there are two registers, named *fEpoch* and *dEpoch*, that control whether an instruction should executed or not. These registers change their value, from False to True and vice versa, only in case of a misprediction of the PC's next value. We have cases of misprediction if the type of the instruction is Branch or Branch

and Link and if the instruction changes the PC's value. Finally the instruction, the PC's value, the value of next instruction's address(PC+4) and the value of the fEpoch are enqueued to a PipelineFIFO, named *fetchToDec* value of the PC is increased by four. Basically the data are enqueued in the FIFO as a structure with three fields.

4.7.3 doDecode rule

This rule has the same enable conditions as the doFetch rule. In this stage the data of the *fetchToDec* PipelineFIFO is being read and the instruction is being decoded.

```

fromFetch = fetchToDec.first;
let pc = fromFetch.pc;
let ppc = fromFetch.ppc;
let dEpochLocal = fromFetch.epoch;
let instr = fromFetch.inst;
let dInst = decode(instr);

```

Then the decoded instruction is along with the data from the previous FIFO are being enqueued to the *decToExec*. The last operation of this rule is to dequeue the data of from the *fetchToDec* FIFO

```

decToExec.enq(Decode2Execpc:pc,ppc:ppc,dInst:dInst,epoch:dEpochLocal );
fetchToDec.deq;

```

4.7.4 doExecute rule

The firing conditions of this rule are the same as the previous two rules. At first the rule reads the data of the *decToExec* FIFO and then checks if the epoch of the instruction is the

same as the epoch of the decode stage. After that and depending on the instruction read the proper registers from the register file. Then the execute function is being called.

```
let eInst = execute(startCPSR, dInst, rnVal, rsVal, rmVal, expc,ppc,startSPSR);
```

The next step, after the function finishes its operation, is to check if the condition flags are true. Only then, the results of the instruction is permanent. According to the type of the instruction some actions might take place.

```
rf.wr(validValue(eInst.rd),eInst.data,startMode);
```

if the instruction wants to store a value to a register, then a write in the register file is performed. In the case the type of the instructions is Store(for all the kinds of Store instructions), then a request is sent to the data memory,to save some data to a specific address that came from the Exec module.

```
dMem.portA.request.put(dMemRequest(True, eInst.addr, storeVal));
```

On the other hand if the type of the instruction suggests a load from the data memory, then the memory receives a request to read data from an address.

```
dMem.portA.request.put(dMemRequest(False, eInst.addr, 0));
```

The flags of the CPSR are changed if the set_flags bit is set.

```
if (isValid(dInst.set_flags) && validValue(dInst.set_flags) == 1)
```

```
begin
```

```
curCPSR[31:28] = eInst.flags;
```

There are some tricky situations for the processor, regarding of some instructions in specific,

that will be discussed below.

Misprediction for the next PC's value When the processor has to execute a branch instruction or another instruction causes the value of the PC to change, then in order to alter the register's value properly, the following are done. At first the mispredict variable is raised during the execution of the instruction. Afterwards the PC's new value is enqueued in a FIFO, named *execRedirect* and the dEpoch register reverses its value. So, in the next cycle the fetch state will understand that a new value has come for the PC, it will instruct the pipeline to kill the instructions that are already in the pipeline and it will fill the pipeline with new instructions after 3-cycles.

Load Instruction Load instructions has some other challenges for the processor. In the case that the type of the instruction is such that the base register (Rn) is required also to be stored, besides the destination register, then we have to perform two stores in the register file. Further ado, due to the latency of the data memory, the data will not be ready for reading and processing until after one cycle. So the processor will have to wait to store the data to the Rd. For this reason the rule doWriteBack was created. If a load instruction suggests a write to the base register, then this write is performed at the execution stage. Simultaneously when the processor realizes that there is a load instruction for execution, it stalls the pipeline for one cycle, until the data from the memory are ready to stored in the register file. In order for the pipeline to stall and the doWriteBack rule to fire properly, the *pendingLoad* register changes its value to True and the data of the *decToExec* PipelineFIFO are not being dequeued. Furthermore the doWriteBack module have to know the rd, rn, the data that would be stored in rn and if there is a misprediction on PC's next value. These data is passed through a FIFO, named *exToMem*.

```
pendingLoad <= True;
```

```
exToMem.enq(Exec2Mem{ data:eInst.data,rd:validValue(eInst.rd),rn:validValue(dInst.rn),
mispredict:eInst.mispredict,lsType:dInst.ldStType });
```

Long multiplication with or without accumulate Similarly, in the case of multiplication, two writes should be made in the register file, one for the lower 32-bits and one for the upper 32-bit of the product. Also four reads of the appropriate registers should be made, thus the three of them will be made at the doExecute rule (due to lack of read ports in the register file) and the last one will be made at the doLongMulRule. Furthermore, in this case when the processor sees a UMULL, SMULL, SMLAL or UMLAL instruction, stalls the pipeline for one cycle. At the execution stage the lower 32-bit (with or without accumulation) are stored in the *RdLo* and during the doLongMull the upper 32-bit of the product are stored in *RdHi*. In order for the doLongMull stage to operate correctly some data are pass through a FIFO from the execute stage. These data are the type of the instruction, the *RdHi*, the upper 32-bit of the product and the flags that ALU raised during the first part of the execution of the instruction.

Changing mode of operation Currently the processor can change its mode of operation, during program execution, with one way. This way is to use the *MSR* and *MRS* instructions. At first with the use of *MRS* instruction, the contents of CPSR can be move to a general purpose register. Then with a series of instructions the last five bits of the register can be alternate. Finally the new value will be moved back to the CPSR with the use of *MSR*. An example of instructions to change the operating mode is as it seems below:

```

MRS      R0, CPSR           ; Copy the PSR
BIC      R0, R0, #&1F      ; Clear the mode bits
ORR      R0, R0, #new_mode ; Set bits for new mode
MSR      CPSR, R0          ; write CPSR back

```

4.7.5 doWriteBack rule

The doWriteBack rule fires if the *pendingLoad*'s value is True. This rule reads the data from the *exToMem* FIFO, receives the response from the data Memory and writes the data to the proper register. But there is a case that one of the rn or the rd registers is the PC. In this case we have a misprediction of the PC's next value. To deal with this situation, the processor follows the same method as before. The new value for the PC is enqueued to a FIFO (*memRedirect*) the dEpoch reverses its value, so the fetch state can handle properly the misprediction. Finally the rule dequeues the data from the *exToMem* FIFO and makes the value of *pendingLoad* register to False, in order for the pipeline to restart.

4.7.6 doLongMul rule

The doLongMul fires if the *pendingLoadMul*'s value is True. This rule reads the data from the *exToMul* FIFO and according to the instruction (with or without accumulation) performs one more addition. It then calculates the final flags for the CPSR, taking into account the temperate flags that the exec module had calculated in the previous cycle. Finally writes the result to the register file, sets the flags if the set_flags bit is set, changes the *pendingLoadMul* register to False and dequeues the data from the *exForMul* FIFO.

Chapter 5

Debugging and Testing

5.1 Debug of the design

The debugging of the project was done by creating the Trace module, using certain functions to display messages and by studying VCD files.

5.1.1 Trace module

In Trace module the following function was created:

```
function Action trace(Fmt fmt) = traceEnabled ? $display($format("[Trace] ") + fmt)
: noAction;
```

The *\$display* invokes the system task to display a literal String (like `printf()` in C, it does formatted output during simulation). The *\$format* is also a display related system task. *\$format* is a value function which returns an object of type `Fmt`. The `Fmt` primitive type provides a representation of arguments to the *\$display* family of system tasks that can be manipulated in BSV code. `Fmt` representations of data objects can be written hierarchically and applied to polymorphic types. Trace function is called in every stage of the pipeline and displays in the console of Bluespec Development Workstation(BDW) the messages.

In order to check if the instruction is decoded appropriately an instance of `fshow`. The `FShow` package defines the typeclass `FShow`. `FShow` includes a single member function, `fshow`. When

applied to an object which is an instance of `FShow`, the `fshow` function returns an object of type `Fmt`.

```
instance FShow#(DecodedInst);  
function Fmt fshow(DecodedInst dInstr);
```

This function was created according to the *decode* functions and displays in the console the type of the instruction the condition flags, the registers that are involved and the operation that barrel shifter has to do. This is the way it is called.

```
trace($format("[Decode State] PC[%d] ", pc)+fshow(dInst));
```

5.2 Testing of the register file

The testing of the register file was done with the creation of a module. The code of the module is shown in the figure below.

Figure 5.1: TestBench the register file

```

1  import RFileModed::*;
2  import Types::*;
3
4  typedef enum {Start, Run} State deriving (Bits, Eq);
5  (* synthesize *)
6  module mkTestBench();
7      Reg#(Int#(32)) cycle <- mkReg(0);
8      Reg#(State) state <- mkReg(Start);
9      RFile rf <- mkRFile;
10     rule start(state == Start);
11         state <= Run;
12     endrule
13     rule countCycle(state == Run);
14         cycle <= cycle + 1;
15     endrule
16     rule r1(cycle <15 );
17         Bit#(32) data = pack(cycle);
18         Bit#(32) ridx = pack(cycle);
19         rf.wr(ridx[3:0],data,usr);
20     endrule
21     rule r2((cycle>15) && (cycle<31));
22         Bit#(32) data = pack(cycle);
23         Bit#(32) ridx = pack(cycle);
24         rf.wr(ridx[3:0],data,svc);
25     endrule
26     rule r3((cycle>31) && (cycle<47));
27         Bit#(32) data = pack(cycle);
28         Bit#(32) ridx = pack(cycle);
29         rf.wr(ridx[3:0],data,irq);
30     endrule
31     rule r4((cycle>47) && (cycle<63));
32         Bit#(32) data = pack(cycle);
33         Bit#(32) ridx = pack(cycle);
34         rf.wr(ridx[3:0],data,abt);
35     endrule
36     rule r5((cycle>63) && (cycle<79));
37         Bit#(32) data = pack(cycle);
38         Bit#(32) ridx = pack(cycle);
39         rf.wr(ridx[3:0],data,und);
40     endrule
41     rule r6((cycle>79) && (cycle<95));
42         Bit#(32) data = pack(cycle);
43         Bit#(32) ridx = pack(cycle);
44         rf.wr(ridx[3:0],data,fiq);
45     endrule
46     rule r7((cycle>95) && (cycle<111));
47         Bit#(32) data = pack(cycle);
48         Bit#(32) ridx = pack(cycle);
49         rf.wr(ridx[3:0],data,sys);
50     endrule
51     rule r8((cycle>111) && (cycle<127));
52         Bit#(32) data = pack(cycle);
53         Bit#(32) ridx = pack(cycle);
54         rf.wr(ridx[3:0],data,fiq);
55     endrule
56     rule r9(cycle==130);
57         $display("_____ Test Bench finished _____");
58         $finish;
59
60     endrule
61 endmodule

```


case that the instruction which came has an immediate as second operand and not a register as in the previous states.

5.4 Testing the processor

To test the design properly a new module was created. In this module the processor's module instantiates along with two registers to control the flow of the program. The code of the TestBench is as follows:

```
typedef enum {Start, Run} State deriving (Bits, Eq);
```

```
(* synthesize *)
```

```
module mkTestBench();
```

```
Proc proc <- mkProcessor;
```

```
Addr addr = 0;
```

```
Reg#(Bit#(32)) cycle <- mkReg(0);
```

```
Reg#(State) state <- mkReg(Start);
```

```
rule start(state == Start);
```

```
proc.hostToCpu(addr);
```

```
state <= Run;
```

```
endrule
```

```
rule countCycle(state == Run);
```

```
cycle <= cycle + 1;
```

```
endrule
```

```
rule halt(state == Run && cycle==200)
```

```
$display("Processor halted");
```

```
$finish;
```

endrule

endmodule

This module consists of three rules. The rule `start` fires if the state is "Start" then it starts the coprocessor and changes the state to "Run". The `countCycle` rule fires when the state is "Run" and he increases the value of cycle register by one. The rule `halt` begins when the state is "Run" and the cycle's value is 200. This value is arbitrary and it depends on how many are the instructions to be executed by the processor.

In order the processor to execute a series of random instructions or a particular program, the Instruction memory must be loaded with a binary file. This file contains on every line an instruction in hexadecimal format. To convert the ARM instructions into hexadecimal format, an online tool was used. The tool can be found in this link : <http://armconverter.com/>

At first some of the instructions was written by hand just to verify that the processor actually works. Then, and since this method is not the most efficient, the ARM gcc-6.3.0 was used. The use of the compiler can be done either from the terminal of a Linux system either from an online tool. The online tool that was used is <https://gcc.godbolt.org/>. Also another tool was used to verify the hexadecimal format of the instructions and to check that the branch instruction will change the PC's value properly. The tool can be found at <https://onlinedisassembler.com/odaweb/> .

The steps to convert a C++ code to raw bytes that the processor are : At first with the use of ARM GCC 6.3.0 for Linux, the code is translated to ARM assembly. For the GCC the next two flags was used:

fomit-frame-pointer

-mcpu=arm7tdmi

The first one tells the tool not to use the frame pointer while creating the assembly instruction and the second one specifies the target device. In continue the assembly code is saved on a .s file. Then using the GNU Arm Embedded Toolchain and the following commands on the terminal, the output is a .bin file on hexadecimal format

```
arm-none-eabi-as -EB -o example.o example.s
arm-none-eabi-ld -EB -Ttext=0x0 -o example.elf example.o
arm-none-eabi-objcopy -O binary example.elf example.bin
```

The first command assembles the file. The second calls the linker of GNU Toolchain. The -Ttext=0x0, specifies that addresses should be assigned to the labels, such that the instructions were starting from address 0x0. Lastly using the objcopy the .bin file is created. This procedure was followed for all the testing codes except Linear search program.

5.4.1 Linear search

The first assembly test code for the processor was written by hand. The program does a linear search to the data memory to find a specific key. If it finds it stores the integer 33 to R3 else the value 66 is stored. The data memory was initialized from a file, so it had some information for the code to search for. The key of the search was on the address 0x07 of the memory. At the end of the program R5 has the position that the key was found and R2 the value of the key. The key in this example was the integer 21. The assembly code is the following

```
mov r0,#0 //base address
mov r1,#9 // size of the array
mov r5,#0 //position counter
mov r3,#21 // key
loop:
```

```

ldr r4,[r0],#1

cmp r4,r3

mov r2,r3

beq end

add r5,r5,#1

cmp r0,r2

bne loop

mov r3,#66

end:

mov r3,#33

```

Figure 5.5: Linear Search console output

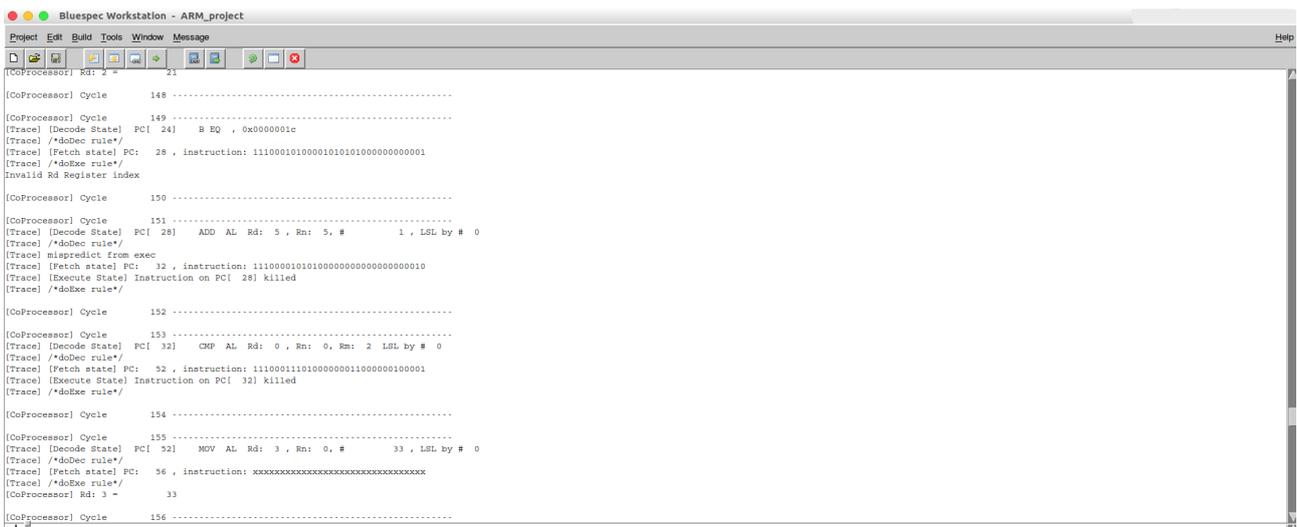


Figure 5.6: Linear Search VCD

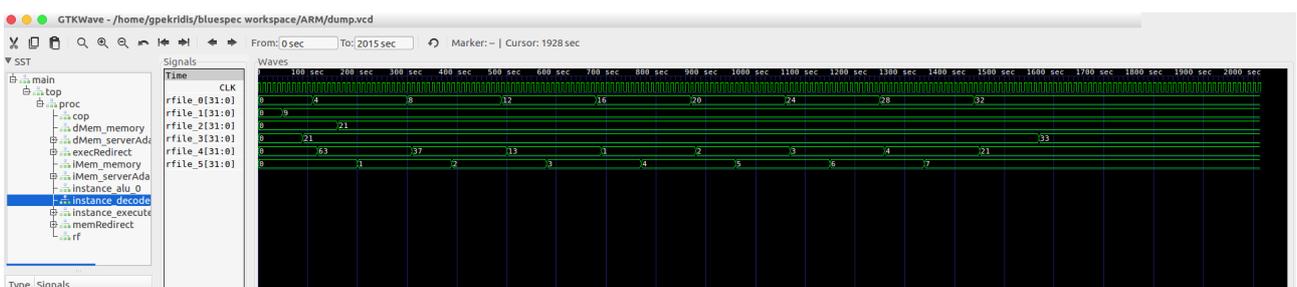


Figure 5.1 presents the output of the console on BDW. In the figure we can see how the debug messages are helping on the verification of the design. The figure 5.2 presents the waveform of the linear search code. Register R0 is on the vector rfile_0[31:0], accordingly all the other registers. As we can see the base register(R0) is increased by four, R1 holds the size of the "table" (the data memory in this case is considered as a table), R2 holds the value of the key, R3 takes the value of the key if the search is successful, R4 holds the value of every element in the search and finally R5 keeps the index of the table.

5.4.2 Largest number among 3

For this test the code to find the largest number among three was written in C++. The C++ and the assembly code are shown in the figure below.

Figure 5.7: Largest number among 3 C++ and assembly code

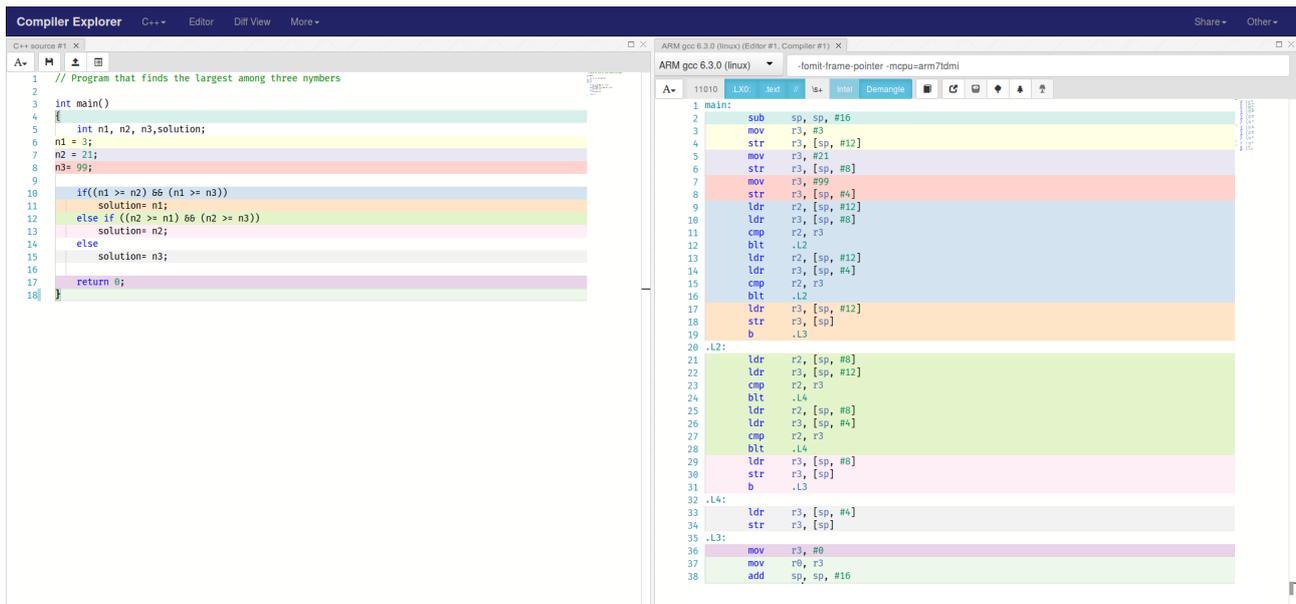
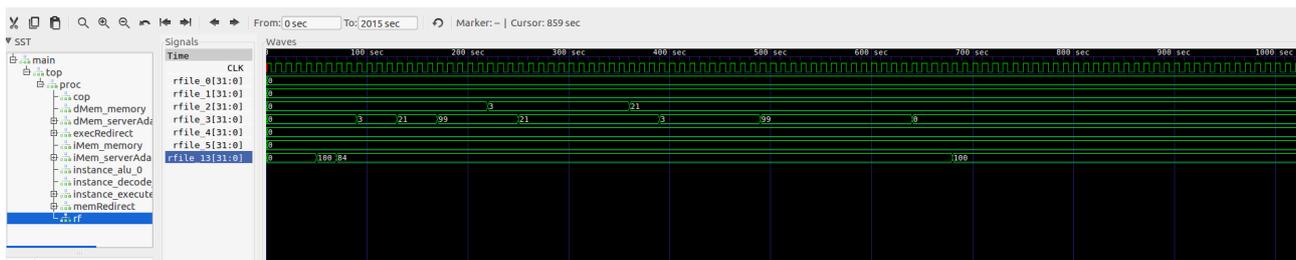


Figure 5.8: Largest number among 3 VCD



At first the stack pointer is initialized at 100. Then the three values are kept in the stack and compares are made between them. The largest number is last stored in R3(in this case 99). After that R3's value becomes zero and the stack pointer is restored.

5.4.3 Factorial

Another program that tested the processor was the calculation of factorial of a certain number(8 in this example). The C++ and the assembly code are shown below.

Figure 5.9: Factorial C++ and assembly code

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code for a factorial function, and the right pane displays the corresponding ARM assembly code generated by GCC 6.3.0.

```

1 int main()
2 {
3     unsigned int n = 8;
4     unsigned long long factorial = 1;
5
6     for(int i = 1; i <=n; ++i)
7     {
8         factorial *= i;
9     }
10
11     return factorial;
12     //Factorial of 8 is 40320;
13 }
14

```

```

ARM gcc 6.3.0 (linux) (Editor #1, Compiler #1)
-fomit-frame-pointer -mcpu=arm7tdmi
11010
1 main:
2     str    r4, [sp, #-4]!
3     sub    sp, sp, #20
4     mov    r3, #8
5     str    r3, [sp]
6     mov    r3, #1
7     mov    r4, #0
8     str    r3, [sp, #8]
9     str    r4, [sp, #12]
10    mov    r3, #1
11    str    r3, [sp, #4]
12    .L3:
13    ldr    r2, [sp, #4]
14    ldr    r3, [sp]
15    cmp    r2, r3
16    bhi    .L2
17    ldr    r3, [sp, #4]
18    mov    r1, r3
19    asr    r2, r1, #31
20    ldr    r3, [sp, #12]
21    mul    r0, r1, r3
22    ldr    r3, [sp, #8]
23    mul    r3, r2, r3
24    add    r0, r0, r3
25    ldr    ip, [sp, #8]
26    umull r3, r4, ip, r1
27    add    r2, r0, r4
28    mov    r4, r2
29    str    r3, [sp, #8]
30    str    r4, [sp, #12]
31    str    r3, [sp, #8]
32    ldr    r3, [sp, #12]
33    ldr    r3, [sp, #4]
34    add    r3, r3, #1
35    str    r3, [sp, #4]
36    b     .L3
37    .L2:
38    ldr    r3, [sp, #8]
39    mov    r0, r3
40    add    sp, sp, #20
41    ldr    r4, [sp], #4

```

The factorial of 8 is:

$8! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 = 40320$. So this is the answer that is expected. From the assembly code we can see that the solution will be stored in R3, then R3's value will become zero and finally the stack pointer will be restored to his initial value(100 in this case)

Figure 5.10: Factorial console output

```

Project Edit Build Tools Window Message
[CoProcessor] Cycle 516 -----
[CoProcessor] Cycle 517 -----
[Trace] [Decode State] PC[ 64] MOV AL Rd: 1 , Rn: 0, Rm: 3 LSL by # 0
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC: 140 , instruction: 11100101100111010011000000001000
[Trace] [Execute State] Instruction on PC[ 64] killed
[Trace] /*doExe rule*/

[CoProcessor] Cycle 518 -----

[CoProcessor] Cycle 519 -----
[Trace] [Decode State] PC[ 140] LDR Rd: 3 [Rn:13, #+ 8 ]
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC: 144 , instruction: 11100011101000000011000000000000
[Trace] /*doExe rule*/
[CoProcessor] Rd: 3 = 84

[CoProcessor] Cycle 520 -----
[CoProcessor] Rd: 3 = 40320

[CoProcessor] Cycle 521 -----
[Trace] /*doWriteBack rule*/
[Trace] [Decode State] PC[ 144] MOV AL Rd: 3 , Rn: 0, # 0 , LSL by # 0
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC: 148 , instruction: 11100001101000000000000000000011

[CoProcessor] Cycle 522 -----
[CoProcessor] Rd: 3 = 0

[CoProcessor] Cycle 523 -----
[Trace] /*doExe rule*/
[Trace] [Decode State] PC[ 148] MOV AL Rd: 0 , Rn: 0, Rm: 3 LSL by # 0
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC: 152 , instruction: 111000101000110111010000000010100

[CoProcessor] Cycle 524 -----
[CoProcessor] Rd: 0 = 0

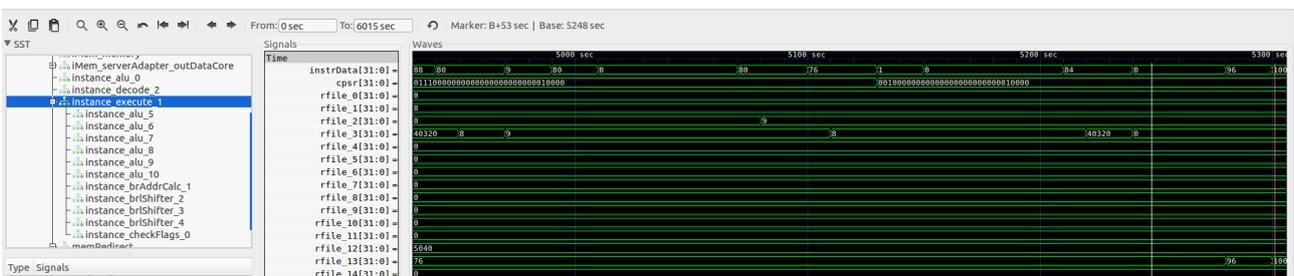
[CoProcessor] Cycle 525 -----
[Trace] /*doExe rule*/
[Trace] [Decode State] PC[ 152] ADD AL Rd: 13 , Rn: 13, # 20 , LSL by # 0
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC: 156 , instruction: 11100100100111010100000000000100

[CoProcessor] Cycle 526 -----
[CoProcessor] Rd:13 = 96

[CoProcessor] Cycle 527 -----

```

Figure 5.11: Factorial VCD output



As we see in the figures above, R3 has the solution of 8! before it turns its value to zero and R13(sp) is restored to 100. The final instructions of the program can be observed also in the

figure 5.6.

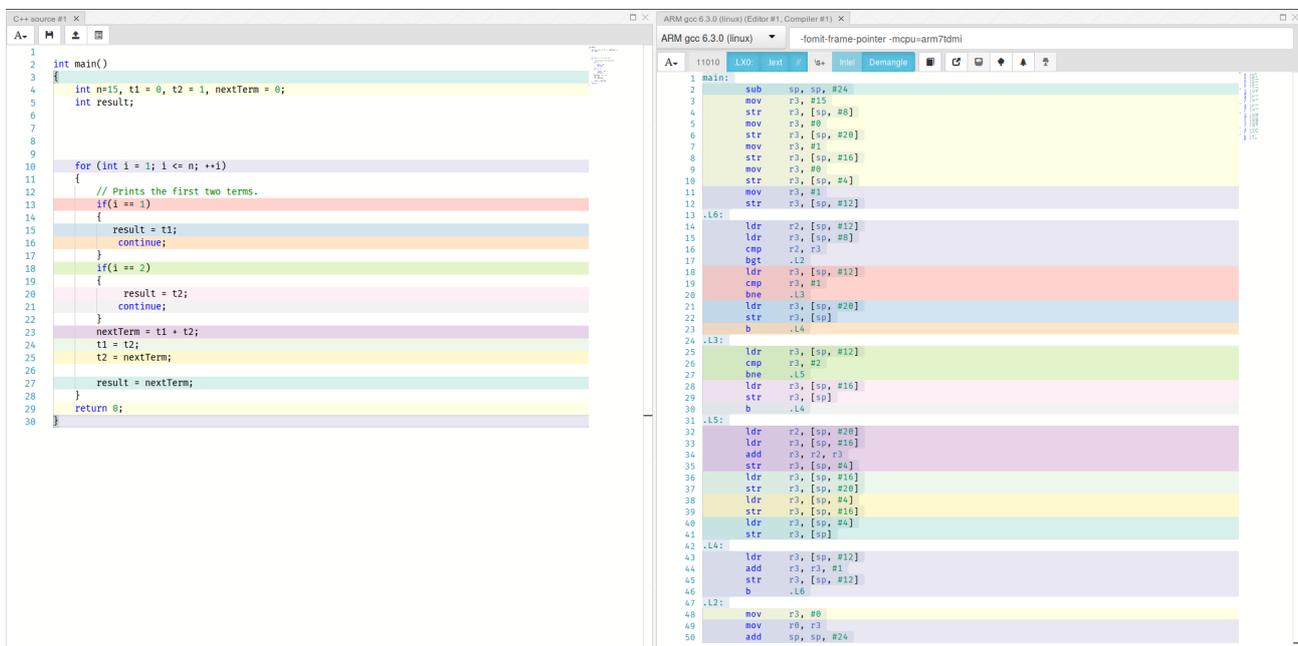
5.4.4 Fibonacci

The next test was the Fibonacci sequence. The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

The next number is found by adding up the two numbers before it. The C++ and the assembly code for the Fibonacci is given below. This code produces the Fibonacci sequence up to n number of terms. In this case n=15, so the expected sequence is:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377.

Figure 5.12: Fibonacci sequence C++ and assembly code



```

1 int main()
2 {
3     int n=15, t1 = 0, t2 = 1, nextTerm = 0;
4     int result;
5
6     for (int i = 1; i <= n; ++i)
7     {
8         // Prints the first two terms.
9         if(i == 1)
10        {
11            result = t1;
12            continue;
13        }
14        if(i == 2)
15        {
16            result = t2;
17            continue;
18        }
19        nextTerm = t1 + t2;
20        t1 = t2;
21        t2 = nextTerm;
22        result = nextTerm;
23    }
24    return 0;
25 }

```

```

ARM gcc 6.3.0 (linux)
-fomit-frame-pointer -mcpu-arm7dmi
1 main:
2     sub sp, sp, #24
3     mov r3, #15
4     str r3, [sp, #8]
5     mov r3, #0
6     str r3, [sp, #20]
7     mov r3, #1
8     str r3, [sp, #16]
9     mov r3, #0
10    str r3, [sp, #4]
11    mov r3, #1
12    str r3, [sp, #12]
13 .L6:
14    ldr r2, [sp, #12]
15    ldr r3, [sp, #8]
16    cmp r2, r3
17    bgt .L2
18    ldr r3, [sp, #12]
19    cmp r3, #1
20    bne .L3
21    ldr r3, [sp, #20]
22    str r3, [sp]
23    b .L4
24 .L3:
25    ldr r3, [sp, #12]
26    cmp r3, #2
27    bne .L5
28    ldr r3, [sp, #16]
29    str r3, [sp]
30    b .L4
31 .L5:
32    ldr r2, [sp, #20]
33    ldr r3, [sp, #16]
34    add r3, r2, r3
35    str r3, [sp, #4]
36    ldr r3, [sp, #16]
37    str r3, [sp, #20]
38    ldr r3, [sp, #4]
39    str r3, [sp, #16]
40    ldr r3, [sp, #4]
41    str r3, [sp]
42 .L4:
43    ldr r3, [sp, #12]
44    add r3, r3, #1
45    str r3, [sp, #12]
46    b .L6
47 .L2:
48    mov r3, #0
49    mov r0, r3
50    add sp, sp, #24

```

As we see in the code the result(the Fibonacci number) is going to be stored on R3.

Figure 5.13: Fibonacci sequence console output

```

Bluespec Workstation - ARM_project
Project Edit Build Tools Window Message Help

[CoProcessor] Cycle      1008 -----
[CoProcessor] Rd: 3 =    377

[CoProcessor] Cycle      1009 -----
[Trace] /*doWriteBack rule*/
[Trace] [Decode State] PC[ 140]   STR Rd: 3 [Rn:13, #+    16 ]
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC:  144 , instruction: 1110010110011101001100000000100

[CoProcessor] Cycle      1010 -----
[CoProcessor] Rd: 3 =     92

[CoProcessor] Cycle      1011 -----
[Trace] /*doExe rule*/
[Trace] [Decode State] PC[ 144]   LDR Rd: 3 [Rn:13, #+     4 ]
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC:  148 , instruction: 1110010110001101001100000000000

[CoProcessor] Cycle      1012 -----
[CoProcessor] Rd: 3 =     80

[CoProcessor] Cycle      1013 -----
[Trace] /*doExe rule*/
[Trace] /*doWriteBack rule*/
[CoProcessor] Rd: 3 =    377

[CoProcessor] Cycle      1014 -----

[CoProcessor] Cycle      1015 -----
[Trace] [Decode State] PC[ 148]   STR Rd: 3 [Rn:13, #+     0 ]
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC:  152 , instruction: 11100101100111010011000000001100
[Trace] /*doExe rule*/
[CoProcessor] Rd: 3 =     76

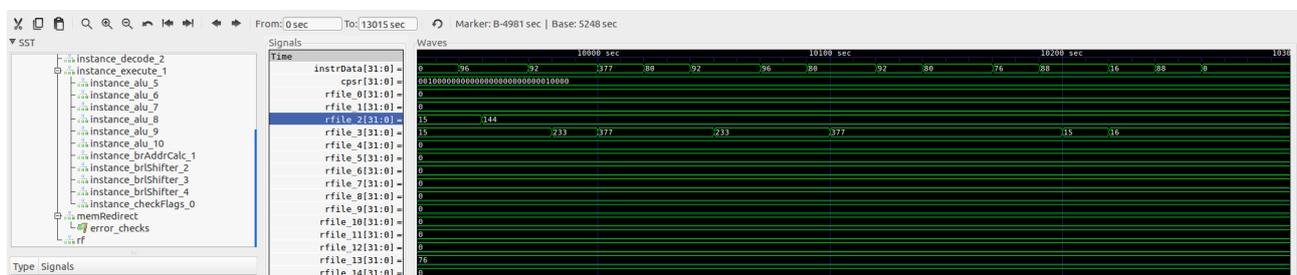
[CoProcessor] Cycle      1016 -----

[CoProcessor] Cycle      1017 -----
[Trace] [Decode State] PC[ 152]   LDR Rd: 3 [Rn:13, #+    12 ]
[Trace] /*doDec rule*/
[Trace] [Fetch state] PC:  156 , instruction: 11100010100000110011000000000001
[Trace] /*doExe rule*/
[CoProcessor] Rd: 3 =     88

[CoProcessor] Cycle      1018 -----
[CoProcessor] Rd: 3 =     15

```

Figure 5.14: Fibonacci sequence VCD output

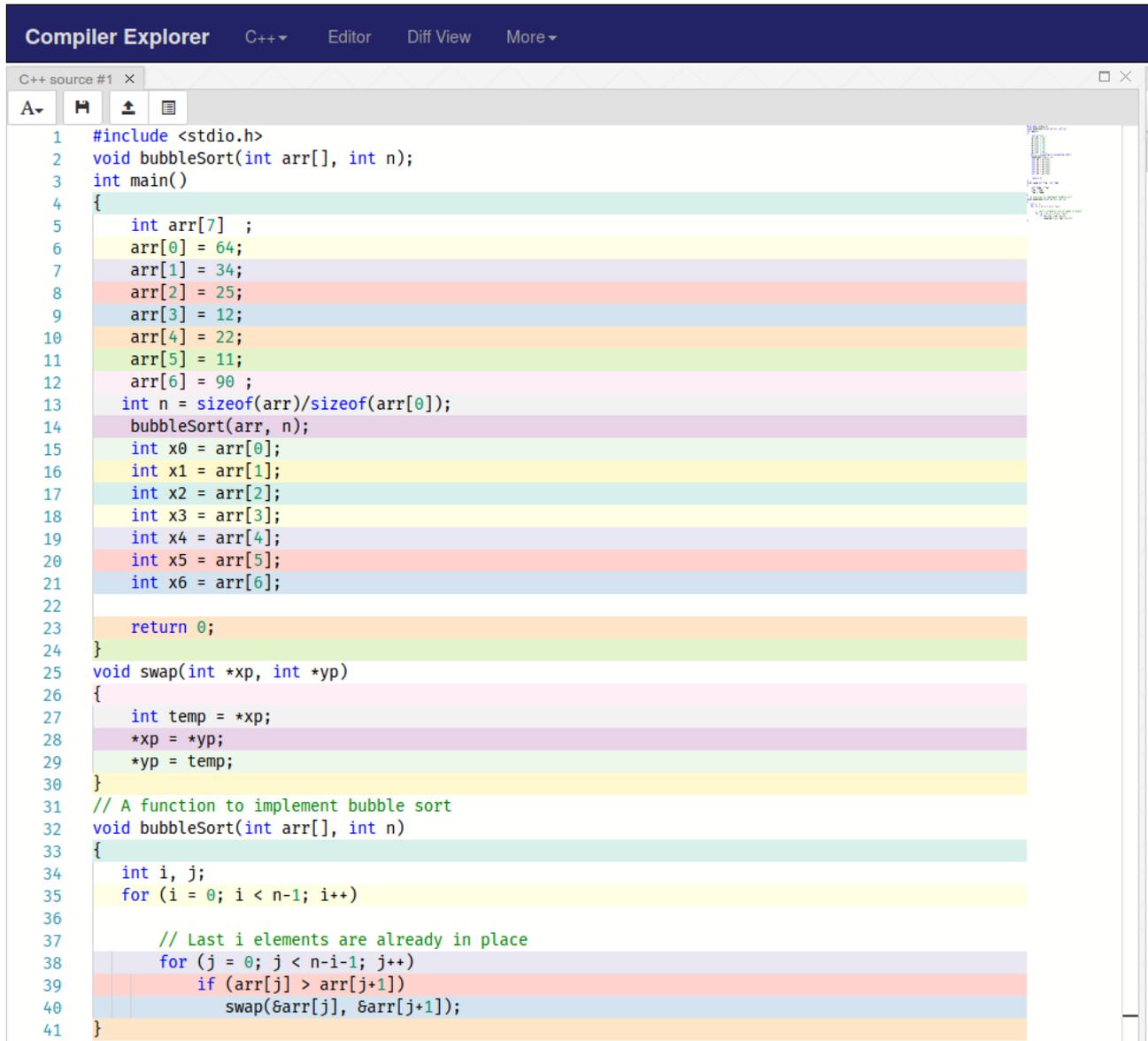


We can see in both figures that the 15th term of the Fibonacci sequence is stored in R3.

5.4.5 Bubble Sort

For this test, the classic algorithm for the Bubble sort was implemented. The code has been implemented in such a way that it is easier to read its results from the assembly code and consequently from the VCD file. The C++ code is as it follows:

Figure 5.15: Bubble sort C++ code



```
1 #include <stdio.h>
2 void bubbleSort(int arr[], int n);
3 int main()
4 {
5     int arr[7] ;
6     arr[0] = 64;
7     arr[1] = 34;
8     arr[2] = 25;
9     arr[3] = 12;
10    arr[4] = 22;
11    arr[5] = 11;
12    arr[6] = 90 ;
13    int n = sizeof(arr)/sizeof(arr[0]);
14    bubbleSort(arr, n);
15    int x0 = arr[0];
16    int x1 = arr[1];
17    int x2 = arr[2];
18    int x3 = arr[3];
19    int x4 = arr[4];
20    int x5 = arr[5];
21    int x6 = arr[6];
22
23    return 0;
24 }
25 void swap(int *xp, int *yp)
26 {
27     int temp = *xp;
28     *xp = *yp;
29     *yp = temp;
30 }
31 // A function to implement bubble sort
32 void bubbleSort(int arr[], int n)
33 {
34     int i, j;
35     for (i = 0; i < n-1; i++)
36
37         // Last i elements are already in place
38         for (j = 0; j < n-i-1; j++)
39             if (arr[j] > arr[j+1])
40                 swap(&arr[j], &arr[j+1]);
41 }
```

Below we can observe the code that the GCC tool produced.

```

1  main:
2
3      mov     sp, #100
4      str     lr, [sp, #-4]!
5      sub     sp, sp, #68
6      mov     r3, #64
7      str     r3, [sp, #4]
8      mov     r3, #34
9      str     r3, [sp, #8]
10     mov     r3, #25
11     str     r3, [sp, #12]
12     mov     r3, #12
13     str     r3, [sp, #16]
14     mov     r3, #22
15     str     r3, [sp, #20]
16     mov     r3, #11
17     str     r3, [sp, #24]
18     mov     r3, #90
19     str     r3, [sp, #28]
20     mov     r3, #7
21     str     r3, [sp, #60]
22     add     r3, sp, #4
23     ldr     r1, [sp, #60]
24     mov     r0, r3
25     bl     bubbleSort
26     ldr     r3, [sp, #4]
27     str     r3, [sp, #56]
28     ldr     r3, [sp, #8]
29     str     r3, [sp, #52]
30     ldr     r3, [sp, #12]
31     str     r3, [sp, #48]
32     ldr     r3, [sp, #16]
33     str     r3, [sp, #44]
34     ldr     r3, [sp, #20]
35     str     r3, [sp, #40]
36     ldr     r3, [sp, #24]
37     str     r3, [sp, #36]
38     ldr     r3, [sp, #28]
39     str     r3, [sp, #32]
40     mov     r3, #0
41     mov     r0, r3
42     add     sp, sp, #68
43     ldr     lr, [sp], #4
44     mov     r5, #66
45     b     stop
46
47     swap:
48     sub     sp, sp, #16
49     str     r0, [sp, #4]
50     str     r1, [sp]
51     ldr     r3, [sp, #4]
52     ldr     r3, [r3]
53     str     r3, [sp, #12]
54     ldr     r3, [sp]

```

Figure 5.16: Bubble short assembly code part 1 Image

```

53     ldr     r2, [r3]
54     ldr     r3, [sp, #4]
55     str     r2, [r3]
56     ldr     r3, [sp]
57     ldr     r2, [sp, #12]
58     str     r2, [r3]
59     mov     r0, r0
60     add     sp, sp, #16
61     mov     pc, lr
62
63     bubbleSort:
64     str     lr, [sp, #-4]!
65     sub     sp, sp, #20
66     str     r0, [sp, #4]
67     str     r1, [sp]
68     mov     r3, #0
69     str     r3, [sp, #12]
70
71     .L9:
72     ldr     r3, [sp]
73     sub     r2, r3, #1
74     ldr     r3, [sp, #12]
75     cmp     r2, r3
76     ble     .L10
77     mov     r3, #0
78     str     r3, [sp, #8]
79
80     .L8:
81     ldr     r2, [sp]
82     ldr     r3, [sp, #12]
83     sub     r3, r2, r3
84     sub     r2, r3, #1
85     ldr     r3, [sp, #8]
86     cmp     r2, r3
87     ble     .L6
88     ldr     r3, [sp, #8]
89     lsl     r3, r3, #2
90     ldr     r2, [sp, #4]
91     add     r3, r2, r3
92     ldr     r2, [r3]
93     add     r3, r3, #1
94     lsl     r3, r3, #2
95     ldr     r1, [sp, #4]
96     add     r3, r1, r3
97     ldr     r3, [r3]
98     cmp     r2, r3
99     ble     .L7
100    ldr     r3, [sp, #8]
101    lsl     r3, r3, #2
102    ldr     r2, [sp, #4]
103    add     r0, r2, r3
104    ldr     r3, [sp, #8]
105    add     r3, r3, #1
106    lsl     r3, r3, #2

```

Figure 5.17: Bubble short assembly code part 2

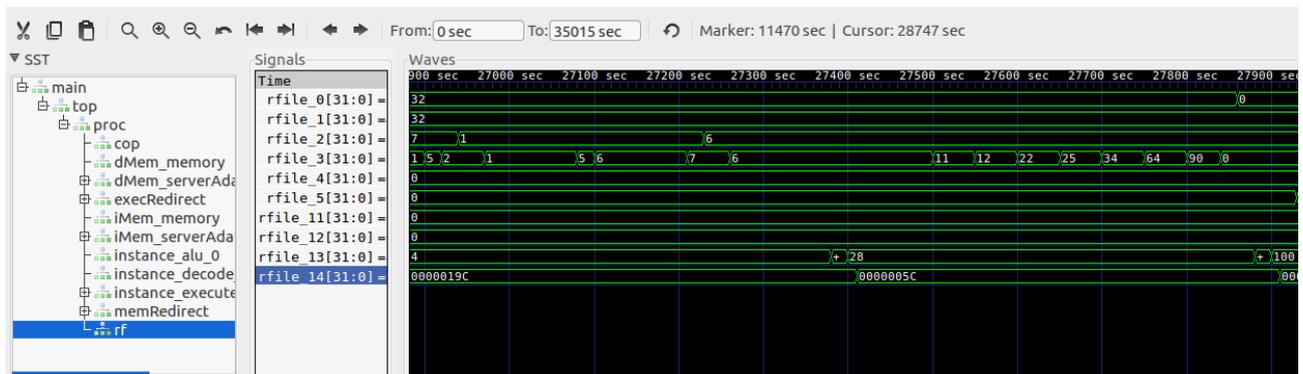
```

106    add     r3, r2, r3
107    mov     r1, r3
108    bl     swap
109
110    .L7:
111    ldr     r3, [sp, #8]
112    add     r3, r3, #1
113    str     r3, [sp, #8]
114    b     .L8
115
116    .L6:
117    ldr     r3, [sp, #12]
118    add     r3, r3, #1
119    str     r3, [sp, #12]
120    b     .L9
121
122    .L10:
123    mov     r0, r0
124    add     sp, sp, #20
125    ldr     lr, [sp], #4
126    mov     pc, lr

```

Figure 5.18: Bubble short assembly code part 3

Figure 5.19: Bubble sort VCD



As we can see in the C++ code the input of the program is an array of 7 integers. The values of the array are 64, 34, 25, 12, 22, 11, 90. At the end of the program we expect the array to be sorted on an ascending order i.e 11, 12, 22, 25, 34, 64, 90. We can see this result on R3 register in the figure 5.15

5.5 Compare with LEON and Piccolo

LEON LEON2 is a 32-bit RISC SPARC V8 compliant architecture, and uses big endian byte ordering as specified in the SPARC V8 reference manual. LEON2 is a synthesizable processor developed by ESA and maintained by Gaisler Research. The processor was originally developed as a fault-tolerant processor for space applications. This report covers the non fault-tolerant version licensed under the GNU LGPL license, which is freely available as a VHDL model from the Gaisler Research website. LEON2 targets both the ASIC and FPGA markets.

Piccolo Piccolo is a 32-bit processor implemented by Bluespec Inc. The architecture of the processor is based on the RV32IM ISA. The features of the free version are:

- 1.9 DMIPS/MHz
- 100 MHz
- 3-stage pipeline
- AXI4-Lite interface
- < 3000 LUTs
- 4KB Instruction & Data caches
- Hardware multiply-divide
- Hardware-based remote GDB

An overview of the three processors is shown in the table below.

Table 5.1: LEON-Piccolo-Our design overview

	Pipeline stages	Clock (MHz)	LUTs
LEON2	5-stage	100	3820-7178
Piccolo	3-stage	100	<3000
Our design	3-stage	100	6330

As we see the LEON2 has a big range for the number of LUTs in the FPGA. This is due to the different types of optimization configuration (Area Optimized, Performance optimized etc.) [8] Piccolo has the least LUTs as it is well optimized by Bluespec Inc. Our design has comparable LUTs with the LEON processor. By studying these results we can conclude that the optimization of our design is feasible, as with this design became a first approach with Bluespec. Our design was synthesized with Vivado 2016.4 and the board that was used is from the Artix-7 family. In the table below we can see how many LUTs each module occupies.

Table 5.2: Detailed allocation of resources

Module name	LUTs	FF
Coprocessor	3	1
Instruction and Data memory	2	2
Instruction and Data memory server adapter	13	6
Register file	1910	960
Decode	66	0
Execute	353	0
execRedirect and memRedirect FIFOs	5	4
Processor (Top module)	6337	1658

From the upper table it is clear that the module with the most LUTs is the top module of the processor. This leads to that the effort of optimizing the code, should start from there.

Chapter 6

Conclusion

6.1 Conclusion of Thesis

This thesis was an attempt to implement a processor in Bluespec also to study if the language is suitable for this kind of work and finally to see what resources occupies in a FPGA the implemented design. The results are that the implementation of a processor is easier than in other HDLs. This is because Bluespec is more like other High Level languages which are used for software(C++,Java) and also provides the ability to design circuits in a more detailed and targeted way. In addition, the level of abstraction may exist in the design, in such a way as to suit the programmer. Another part that is important is that the simulation time is reduced dramatically with Bluesim over a Verilog simulator. The total amount of code lines for the design is 2946. One more benefit of Bluespec is that its learning curve is about one to two months for a person with some programming skills. From the experience that was gained through the process of this thesis. I would also like to mention that the time that I needed to produce some code which implements a simple version of the processor, was 3 to 4 weeks. From this point onwards, due to the complexity of the design, the time that it was needed to produce the rest of modules of the processor, grew exponentially.

6.2 Future Work

- Optimize the code of the 3-stage pipeline processor and see if this has as a result to get higher clock frequency and fewer LUTs on the FPGA.
- Design a processor with 5-stage pipeline and data forwarding.
- Expand the instruction set to another version(ARMv4T, ARMv5T etc).
- Include instruction and data caches.
- Implement a branch prediction unit
- Study the tool to find an optimal way to design processors in Bluespec
- Implement some peripherals for the processor(e.g Debug and Support Unit)
- Implement a multicore processor
- Power management of the design

Bibliography

1. "Bluespec TM SystemVerilog Reference Guide" Revision: 30 July 2014
2. Rishiyur S. Nikhil and Kathy R. Czeck, "BSV by Example" 2010
3. ARM Architecture Reference Manual
4. ARM7TDMI-S Data Sheet
5. Arvind, Rishiyur S. Nikhil ,Joel S. Emer 3 , Murali Vijayaraghavan:"Computer Architecture: A Constructive Approach Using Executable and Synthesizable Specifications" December 2012 [Online]Available:http://csg.csail.mit.edu/6.375/6.375_2016_www/resources/a
6. Nirav Hemant Dave,"Designing a Processor in Bluespec",January 2005
7. Bluespec Testing Results: Comparing RTL Tool Output to Hand-Designed RTL, <http://bluespec.com> 2004
8. D.Mattsson M.Christensson "Evaluation of synthesizable CPU cores" December 2004 [Online]Available:http://www.gaisler.com/doc/Evaluation_of_synthesizable_CPU_cores.pdf
9. <http://wiki.bluespec.com/>
10. <http://infocenter.arm.com>